Checkpoint/rollback vs causally-consistent reversibility

Martin Vassor, Jean-Bernard Stefani

Checkpoint/rollback vs causally-consistent reversibility

Martin Vassor and Jean-Bernard Stefani

Univ. Grenoble-Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. This paper analyzes the relationship between a distributed checkpoint/rollback scheme based on causal logging, called Manetho, and a reversible concurrent model of computation, based on the π-calculus with imperative rollback developed by Lanese et al. in [**?**]. We show a rather tight relationship between rollback based on causal logging as performed in Manetho and the rollback algorithm underlying the calculus in [**?**]. Our main result is that the latter can faithfully simulate Manetho, where the notion of simulation we use is that of weak barbed simulation, and that the converse only holds if possible rollbacks in are restricted.

1 Introduction

Motivations. Undo capabilities constitute a key and early example of reversibility ideas in languages and systems [**?**]. Checkpoint/rollback schemes in distributed systems [**?**] constitute prime examples of such capabilities and of their application to the construction of fault-tolerant systems. There are certainly distinctions to be made between complete reversibility and checkpoint/rollback schemes, if only in terms of the granularity of undone computations, and in terms of the space/time trade-offs that can be made, as illustrated in [**?**] that advocates the use of a reversible language for high performance computing applications. However, one can ask what is the exact relationship between the checkpoint/rollback schemes that have been proposed in the literature, and concurrent reversible models of computation which have been proposed in the past fifteen years, especially the causally consistent ones that have been proposed following Danos and Krivine's work on Reversible CCS [**?**]. More specifically, one can ask what is the relationship between checkpoint/rollback distributed algorithms based on causal logging [**?**], and the distributed algorithms which are implicit in the semantics of reversible concurrent languages such as in the low-level semantics of the reversible higher-order calculus studied in [**?**]. The question is particularly relevant in these cases, because both exploit causal relations between events in distributed computations. But are these relations identical, and how do associated algorithms compare? Comparing causal relations in concurrent models is in itself not trivial. For instance, recent works on concurrent reversible calculi, e.g. [**?**,**?**] show that even when dealing with the same model of computation, causal information can be captured in subtly different ways. To the best of our knowledge, the question of the relationship between checkpoint/rollback and reversible models of computation has not been tackled the literature. This paper

aims to do so. Beyond the gain in knowledge, we think this can point the way to useful extensions to checkpoint/rollback schemes in order to handle finer grain undo actions and to deal with dynamic environments, where processes can be freely created and deleted during execution.

Approach. To perform our analysis, we have chosen to compare the MANETHO algorithm [**?**], with the low-level semantics of the roll-π calculus, which in effect specifies a rollback algorithm that exploits causal information gathered during forward computations. MANETHO is interesting because it constitutes a representative algorithm of so-called causal logging distributed checkpoint/rollback schemes [**?**], and because it combines nicely with replication mechanisms for fault-tolerance. The analysis proceeds as follows. We first define a language named SCL (which stands for Stable Causal Logging, i.e. causal logging with a stable support for checkpoints), which operational semantics formalizes MANETHO operational semantics. We prove that this language has sound rollback semantics, meaning that MANETHO correctly performs its intended rollback. We then define a variant of the roll-π calculus [**?**,**?**] to obtain an intermediate language that is more suitable for the comparison with MANETHO, and in particular that exhibits the same forms of communication and failures as assumed by MANETHO. In this language, named lr-π, a program comprises a fixed number of parallel processes, communication channels are statically allocated, communication between processes is via asynchronous messages (whose delivery may be delayed indefinitely), and processes may crash fail at any time. Finally, we study the operational correspondence between SCL and lr-π, using standard simulation techniques.

Contributions We show a rather tight relationship between rollback based on causal logging as performed in MANETHO and the rollback algorithm of roll-π. Our main result is that lr-π can simulate SCL (hence, so does the regular roll-π), where the notion of simulation we use is that of weak barbed simulation [**?**], and that the converse only holds for a weaker version of lr-π, where possible rollbacks are restricted. This shows that the causal information captured by MANETHO is essentially the same as that captured by roll-π, and that the difference between the two schemes lies in the capabilities of the rollback operations, which are limited in MANETHO (as in most checkpoint/rollback schemes) to rolling back to the last checkpointed state.

Outline. The paper is organized as follows. Section **??** briefly presents MANETHO, formally defines SCL and proves the soundness of rollback in SCL. Section **??** introduces the lr-π calculus and shows that it can be faithfully encoded in the roll-π) calculus. Section **??** presents our main result. Section **??** discusses related work. Section **??** concludes the paper. Due to the size limit, proofs of our results are not presented in the paper, but they are available online [**?**].

2 Formalizing Manetho

2.1 Manetho

MANETHO [?] is a checkpoint-rollback protocol that allows a fixed number of processes, that communicate via asynchronous message passing, to tolerate process crash failures[1]. To achieve this result, each time a non-deterministic event happens, the concerned process enters a new *state interval*. Processes keep track of the current state interval causal dependencies in an *antecedence graph*. Processes can take checkpoints and, upon failure, the failed process rolls back to a causally consistent state from this checkpoint. For instance, in the execution in Figure **??**, at some point, process p receives a message m_3 from process q. Since delivering a message is a non-deterministic event, p enters a new state interval si_p^1. When sending a message, the sender piggybacks its antecedence graph in the message. This allows the receiver to update its own antecedence graph. For instance, in Figure **??**, when q sends message m_3 to p, it piggybacks the antecedence graph shown in Figure **??**, which p uses to update its own antecedence graph, by merging them, resulting in the antecedence graph shown in Figure **??**. Moreover, when a process sends a message, it keeps a local copy of the message.

When a failure occurs, for instance process q in Figure **??**, the process recovers from its last checkpoint (for instance checkpoint c_q^1). Other processes can inform the recovering process of its last known state interval. In the example, process q sent a message m_3 to process p when it was in state interval si_q^2. Hence, the antecedence graph of the state interval si_q^2 is a subgraph of the antecedence graph of state interval si_p^1: process p can then retransmit it to q in an out of band message (the red message from p to q in Figure **??**). With its antecedence graph recovered, the process can replay the message sequence to recover its last state interval globally known, by asking for copies of the received messages (message m_2'). Notice that the recovering process does not resend its message (internally, it only replays the sending event without actually sending the message in order to keep track of message counter values).

Notice that during the recovery, only the recovering process changes: apart from efficiency considerations, this is to ensure MANETHO processes that use checkpoint/rollback can coexist with replicated processes.

2.2 Formalization

We formalize MANETHO processes by means of a small language of configurations, called SCL. Following the original description of MANETHO [?], we model

[1] The description of the MANETHO protocol differs slightly between the publication [?] and Elnozahy's PhD thesis [?]. In particular, the latter involves a coordinating checkpointing scheme, which is not the case in the former. For the sake of simplicity, in this paper we follow the description in [?]. Checkpoint coordination in any case is not necessary for the correct operation of the recovery process in a causal logging checkpoint/rollback scheme. In [?] it is essentially used to simplify the garbage collection of recovery information.

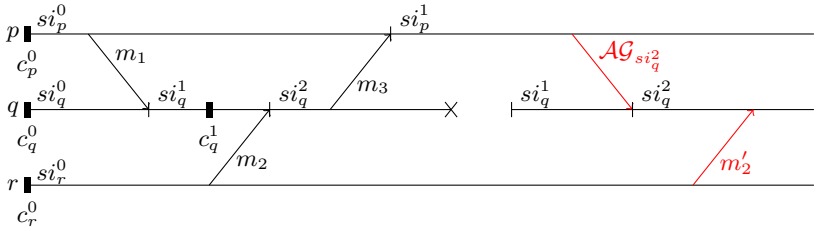

Fig. 1. Message exchanges, failure and recovery.

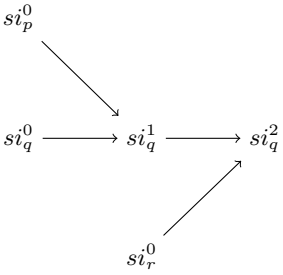

Fig. 2. Antecedence graph of q piggybacked with m_3

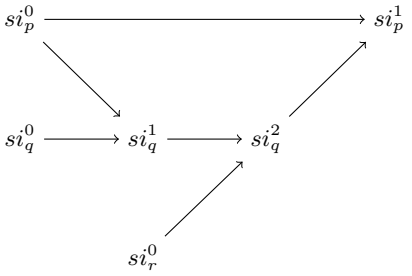

Fig. 3. Antecedence graph of p after merging the antecedence graph piggybacked in m_3.

a configuration $\mathcal{C}$ as a tuple of three elements: (i) a set of processes $\mathcal{M}$; (ii) a set of messages $\mathcal{L}$; (iii) a set of checkpoints $\mathcal{K}$. Checkpoints are just copies of processes, together with a checkpoint identifier.

The processes themselves are composed of a program $\mathcal{P}$, an antecedence graph $\mathcal{AG}$, and a record of sent messages $\mathcal{R}$. They also contain a process identifier k (the set of process identifiers is noted $\mathbb{P}$) and three counters si, inc, ssn which respectively record the state interval, the number of incarnations (i.e. the number of times the process has failed and recovered) and the number of messages sent. Originally, in MANETHO, a *receive counter* is also present. We do not include it in our formalisation, for it can be retrieved from the antecedence graph (the number of messages received is the number of tree merges). We can also retrieve which message triggered each state interval (this information is encoded in each antecedence graph node with a special case for initial state intervals). Finally, processes are decorated with a *mark* which is used to separate the regular evolution of processes from their recovery.

The complete grammar of configurations is provided in Figure **??**, where an empty set is denoted by ε. In the term $\texttt{receive}(source, X) \cdot \mathcal{P}$, $source$ and X are bound in $\mathcal{P}$. In the term $\texttt{send}(dest, \mathcal{P}) \cdot \mathcal{Q}$, $dest$ is a process id, constant or

variable, and $\mathcal{P}$ is a closed process. An antecedence graph $\mathcal{AG}$ is a graph whose nodes are tuples $\langle si, ssn \rangle$.

$$
\begin{array}{lll}
\mathcal{C} & ::= \langle \mathcal{M}, \mathcal{L}, \mathcal{K} \rangle & \textit{SCL configuration} \\
\mathcal{M} & ::= \mathcal{T} \parallel \mathcal{M} \quad | \quad \mathcal{T} & \textit{(Parallel) processes} \\
\mathcal{T} & ::= \langle k, \mathcal{P}, \mathcal{AG}, si, inc, ssn, \mathcal{R} \rangle^{mark} & \textit{SCL process} \\
\mathcal{P}, \mathcal{Q} & ::= 0 \quad | \quad \bot & \textit{Empty program, Failed program} \\
 & \quad | \; X & \textit{Variable} \\
 & \quad | \; \texttt{send}(dest, \mathcal{P}) \cdot \mathcal{Q} & \textit{Send message} \\
 & \quad | \; \texttt{receive}(source, X) \cdot \mathcal{P} & \textit{Receive message} \\
\mathcal{L} & ::= \langle src, dst, ssn, \mathcal{AG}_s, \mathcal{P}, inc \rangle :: \mathcal{L} \quad | \quad \varepsilon & \textit{Set of messages} \\
\mathcal{R} & ::= \langle src, ssn, Q \rangle :: \mathcal{R} \quad | \quad \varepsilon & \textit{Set of sent messages} \\
\mathcal{K} & ::= \langle cid, k, \mathcal{P}, \mathcal{AG}, si, inc, ssn, \mathcal{R} \rangle :: \mathcal{K} \quad | \quad \varepsilon & \textit{Set of checkpoints}
\end{array}
$$

Fig. 4. SCL syntax

Notice that SCL is a higher-order language, where processes exchange messages which carry programs. Although strictly not necessary to model MANETHO configurations, this allows us to accommodate very simply processes with unbounded executions. Consider for instance the following configuration C:

$$\langle \langle k_0, P, \mathcal{AG}_0, si_0, inc_0, ssn_0, R_0 \rangle \parallel \langle k_1, Q, \mathcal{AG}_1, si_1, inc_1, ssn_1, R_1 \rangle, \mathcal{L}, \mathcal{K} \rangle$$

with $Q = \texttt{receive}(source, X) \cdot \texttt{send}(k_0, \texttt{send}(k_1, X) \cdot \texttt{receive}(source, Y) \cdot Y) \cdot X$ and $P = \texttt{send}(k_1, Q) \cdot \texttt{receive}(source, Y) \cdot Y$. This configuration evolves into itself after the exchange of two messages, from k_0 to k_1 and from k_1 to k_0.

For the sake of conciseness, we use the notation $inc(k)$ to denote the incarnation number of the process with process id k.

Starting and Correct Configurations We now define the set of starting configurations: $\mathbb{C}^s_{\text{SCL}}$. A configuration $\mathcal{C}_s = \langle \mathcal{M}, \mathcal{L}, \mathcal{K} \rangle$ is said to be starting if and only if:

- all X and *dest* are bound;
- T_i are not marked;
- there is no pending message: $\mathcal{L} = \emptyset$;
- the set of checkpoint contains a checkpoint of the initial state of each process: $\mathcal{K} = \bigcup_{k \in \mathbb{P}} \{\langle cid^0_k, k, P_k, \mathcal{AG}_k, si_k, inc^0_k, ssn^0_k \rangle\}$;
- processes are not crashed: $\forall T_i \in \mathcal{M} \cdot T_i \neq \langle k, \bot, \mathcal{AG}, si, inc, ssn, R \rangle$;
- there is no causal dependency between processes, i.e. the antecedence graph of each process is a single vertice: $\forall \langle k, P, \mathcal{AG}, inc, ssn, R \rangle \in \mathcal{M} \cdot \mathcal{AG} = \texttt{root}(\mathcal{AG})$.

We also define correct configurations ($\mathbb{C}_{\text{SCL}}$) which are configurations $\mathcal{C}$ such that there exists a starting configuration $\mathcal{C}_s$ and $\mathcal{C}_s \rightarrow^\star \mathcal{C}$ (with $\rightarrow^\star$ the reflexive and transitive closure of $\rightarrow$ defined hereafter).

Operational Semantics The operational semantics of SCL is defined by means of a reduction relation between configurations, noted $\rightarrow$. The transition relation is the union of two relations: a forward relation, noted $\twoheadrightarrow$, that corresponds to normal process execution, and a rollback relation, noted $\rightsquigarrow$, that implements process recovery. Our reduction relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined as the smallest relations defined by a set of inference rules. For space reasons, we do not present all the inference rules but only a relevant sample.

To define the reduction relations, we rely on the usual functions and predicates: $\texttt{succ}()$, $\texttt{pred}()$ on natural numbers, $\cup$, $\cap$, $\in$, etc. on sets, as well as a parallel substitution that operates on free variables: $P\{^{a_1,\ldots,a_n}/_{b_1,\ldots,b_n}\}$ substitutes each b_i with a_i in P. The substitution is performed in parallel, hence $P\{^{a_1}/_{b_1}\}\{^{a_2}/_{b_2}\} \neq P\{^{a_1,a_2}/_{b_1,b_2}\}$[2]. Concerning antecedence graphs, we use a merge operation (noted $\mathcal{AG}_1 \cup_t \mathcal{AG}_2$) between two graphs, which simply creates a new common ancestor t for the roots of the two graphs to be merged, here $\mathcal{AG}_1$ and $\mathcal{AG}_2$.

Forward Rules. We have six forward reduction rules: a rule to *send* a message from one process to another, a rule for a process to *receive* a message, a rule for a process to *lose* a message, a rule to set a new *checkpoint*, a rule for a process to *idle*, and finally a rule corresponding to a process *failure*. For instance, the rule for receiving a message is defined as follows:

$$\text{S.RECEIVE}\ \frac{\begin{array}{c}\mathcal{AG}' = \mathcal{AG} \cup_{\langle \texttt{succ}(si), ssn_0\rangle} \mathcal{AG}_0 \\ \mathcal{L}' = \mathcal{L}\backslash\{\langle k_0, k, ssn_0, \mathcal{AG}_0, Q, inc_0\rangle\} \qquad inc(k_0) = inc_0\end{array}}{\begin{array}{l}\langle\langle k, \texttt{receive}(source, X) \cdot P, \mathcal{AG}, si, inc, ssn, R\rangle \parallel \mathcal{M}, \mathcal{L}, \mathcal{K}\rangle \\ \twoheadrightarrow \langle\langle k, P\{^{k_0,Q}/_{source,X}\}, \mathcal{AG}', \texttt{succ}(si), inc, ssn, R\rangle \parallel \mathcal{M}, \mathcal{L}', \mathcal{K}\rangle\end{array}}$$

In this rule, process k receives a message Q with number ssn_0 and antecedence graph $\mathcal{AG}_0$ from process k_0. The antecedence graph $\mathcal{AG}$ of process k is updated to $\mathcal{AG} \cup_{\langle \texttt{succ}(si), ssn_0\rangle} \mathcal{AG}_0$. Notice that the condition $inc(k_0) = inc_0$ in the rule premises is non local. This is a simplification from the original MANETHO protocol, in which processes maintain a local vector containing incarnation numbers of all processes which is updated by a message broadcast to all other processes following a process recovery, and the condition $inc(k_0) = inc_0$ corresponds to a local look-up at the vector. For the sake of simplicity, we chose not to model this part of the protocol, which is of no relevance for our simulation results.

The rule for process failure is defined as follows:

$$\text{S.FAIL}\ \langle\langle k, P, \mathcal{AG}, si, inc, ssn, R\rangle \parallel M, \mathcal{L}, \mathcal{K}\rangle \twoheadrightarrow \langle\langle k, \bot, \mathcal{AG}, si, inc, ssn, R\rangle, \mathcal{L}, \mathcal{K}\rangle$$

With this rule, any process which is not recovering (i.e. any process without a mark) can fail (i.e. the program is replaced by $\bot$). This rule does not need any additional condition since processes can fail at any time.

[2] In particular: $X\{^{Y}/_{X}\}\{^{Z}/_{Y}\} = Z$ and $X\{^{Y,Z}/_{X,Y}\} = Y$.

Rollback Rules. Rollback is done in three steps: restarting the process from its last checkpoint (one rule *initialise checkpoint*), retrieving the antecedence graph from other processes (two rules: one for the *antecedence graph reconstitution* and one for *ending the antecedence graph reconstitution*) and finally, replaying locally message exchanges until the last state interval of the received tree is reached (three rules: one to *replay messages sent*, one to *replay messages delivered* and the last to *end the replay sequence*). We thus have six reduction rules implementing rollback. In the following, we only give full details on some of them.

The first step of the recovery is to re-instantiate the process from its last checkpoint:

$$\textsc{S.roll}\ \frac{\begin{array}{c}\langle cid_r, k, P_r, \mathcal{AG}_r, si_r, ssn_r, R_r\rangle \in \mathcal{K} \\ cid_r \text{ biggest checkpoint id of process } k.\end{array}}{\begin{array}{l}\langle\langle k, \bot, \mathcal{AG}, si, inc, ssn, R\rangle \parallel M, \mathcal{L}, \mathcal{K}\rangle \\ \rightsquigarrow \langle\langle k, P_r, \mathcal{AG}_r, si_r, \mathtt{succ}(inc), ssn_r, R_r\rangle^{\bullet} \parallel M, \mathcal{L}, \mathcal{K}\rangle\end{array}}$$

Notice that only *inc* is preserved during this rule, and the other fields are recovered from the checkpoint record. This is in line with Manetho, which assumes that both checkpoint and incarnation number are held in stable storage in order to survive a process crash.

Then, we can retrieve the new antecedence graph from other processes. To do that, each other process sends the biggest subtree of their antecedence graph which root belongs to the failed process state intervals. The failed process can then rebuild a new antecedence graph consistent with other processes using the rule S.getAG below:

$$\frac{\mathcal{AG}'_1 \subset \mathcal{AG}_1 \qquad root_{\mathcal{AG}'_1} \text{ is the biggest } si \text{ of } k_0 \text{ in } \mathcal{AG}_1 \qquad root_{\mathcal{AG}'_1} > root_{\mathcal{AG}_0}}{\begin{array}{l}\langle\langle k_0, P_0, \mathcal{AG}_0, si_0, inc_0, ssn_0, R_0\rangle^{\bullet} \parallel \langle k_1, P_1, \mathcal{AG}_1, si_1, inc_1, ssn_1, R_1\rangle \parallel M, \mathcal{L}, \mathcal{K}\rangle \\ \rightsquigarrow \\ \langle\langle k_0, P_0, \mathcal{AG}'_1, si_0, inc_0, ssn_0, R_0\rangle^{\bullet} \parallel \langle k_1, P_1, \mathcal{AG}_1, si_1, inc_1, ssn_1, R_1\rangle \parallel M, \mathcal{L}, \mathcal{K}\rangle\end{array}}$$

This ends when the reconstructed antecedence graph cannot be augmented anymore (rule S.endAG not shown, which changes the mark of the process under recovery from $\bullet$ to $\circ$).

The new antecedence graph is consistent with other processes, but the current state of the recovering process does not match the antecedence graph. Hence, we have to simulate locally the messages sent and received (rules S.replay.send and S.replay.deliver). To replay a delivery, the process simply gathers a saved copy of the message from the sender. To replay a send, it updates its memory, but does not send anything (to avoid double delivery). For instance, the rule S.replay.send is the following:

$$\frac{si_0 \leq root_{\mathcal{AG}_0} \qquad R'_0 = R_0 \cup \langle k_0, ssn_0, Q\rangle}{\begin{array}{l}\langle\langle k_0, \mathtt{send}(k_1, Q) \cdot P_0, \mathcal{AG}_0, si_0, inc_0, ssn_0, R_0\rangle^{\circ} \parallel \mathcal{M}, \mathcal{L}, \mathcal{K}\rangle \\ \rightsquigarrow \langle\langle k_0, P_0, \mathcal{AG}_0, si_0, inc_0, \mathtt{succ}(ssn_0), R'_0\rangle^{\circ} \parallel \mathcal{M}, \mathcal{L}, \mathcal{K}\rangle\end{array}}$$

Once the message is about to enter a new state interval that is not in the recovering antecedence graph, the rollback sequence ends: the end recovery rule S.replay.stop (not shown) simply erases the mark.

2.3 Rollback Soundness

We now show that SCL has sound rollback semantics. This in itself is not surprising since the MANETHO recovery process was already proven correct in [**?**,**?**]. Nonetheless, our proof method differs from that in [**?**,**?**]. The original MANETHO semantics is described in pseudo-code while our formalization of MANETHO configurations and their operational semantics uses reduction rules. Hence, our proofs are based on case studies on the reduction relation, which is a step closer to a computer assisted verification of its correctness.

Definition 1 (Execution and recovery sequence). *Given a set of configurations* $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$*, an* execution *is a sequence of reductions of the form* $\mathcal{C}_1 \to \ldots \to \mathcal{C}_n$*. A* recovery sequence *for a process* k *in a configuration* $\mathcal{C}_\perp$ *is the shortest execution* $\mathcal{C}_\perp \to^\star \mathcal{C}$ *such that process* k *is failed in* $\mathcal{C}_\perp$ *and that it has no mark in* $\mathcal{C}$*.*

Definition 2 (Soundness of recovery sequence). *Let* $\mathcal{C}_1 \to^\star \mathcal{C}_2 \to^\star \mathcal{C}_3$ *be an execution, with* $\mathcal{C}_2 \to^\star \mathcal{C}_3$ *a recovery sequence for a process* k*.* $\mathcal{C}_2 \to^\star \mathcal{C}_3$ *is sound if and only if there exists a configuration* $\mathcal{C}'_3$ *such that* $\mathcal{C}_1 \to^\star \mathcal{C}'_3$ *with* $\mathcal{C}_3$ *and* $\mathcal{C}'_3$ *being identical except for the incarnation number of process* k*.*

Theorem 1 (Rollback soundness). *If* $\mathcal{C}_1$ *is a correct configuration and* $\mathcal{C}_1 \to^\star \mathcal{C}_2$ *is a recovery sequence for some process* k *in* $\mathcal{C}_1$*, then* $\mathcal{C}_1 \to^\star \mathcal{C}_2$ *is sound.*

3 The lr-π Language

We now describe the lr-π language. This language is an intermediate language between the roll-π calculus [**?**] and MANETHO. The lr-π language is a higher order reversible π calculus in which processes are located and messages are exchanged asynchronously with continuations. The syntax is similar to roll-π's syntax, except that (i) there is no name creation; (ii) processes are identified by a *tag* (to record causal dependencies between events) and a *label* (which acts as a process identifier); (iii) there are continuations after sending a message; (iv) the number of parallel processes in an lr-π configuration is fixed and does not change during execution.

3.1 Syntax

A configuration is a list of processes running in parallel. Each process is identified by a location λ and a tag k, which serves the same purpose of tracking causality as the tags in [**?**]. We are given a set $\mathbb{X}$ of variables (elements among x, y, ...), a set $\mathbb{L}_v$ of location variables (l_1, l_2, ...) and a set $\mathbb{L}_c$ of location constants (λ_1, λ_2, ...). We let $\mathbb{L}$ denote $\mathbb{L}_v \cup \mathbb{L}_c$, with elements among λ, λ', ... The action of sending program $\mathcal{P}$ to a process λ creates a *message* at that location: $k_i, \lambda : \lambda_j[\mathcal{P}]$. Failed processes are written with the symbol $\perp$.

The lr-π constructs are similar to roll-π ones: upon delivery of a message, a memory is created to keep track of the evolution of the configuration in order to

reverse deliveries. The tag of the memory corresponds to the tag of the resulting process. For instance, the configuration $k_0, \lambda_1 : x \triangleright_l 0 \parallel k', \lambda_1 : \lambda_2[0]$ reduces to $k_1, \lambda_1 : 0 \parallel [k_0, \lambda_1 : x \triangleright_l 0 \parallel k', \lambda_1 : \lambda_2[0]; k_1]$. We know from the tags that the process $k_1, \lambda_1 : 0$ results from the delivery kept in the memory $[\ldots; k_1]$. Finally, frozen variants of processes are used during backward reductions to mark processes that should be reverted.

The complete grammar is provided in Figure **??**.

$\mathcal{C}$	$::= \mathcal{M} \parallel \mathcal{C} \quad \mid \quad \varepsilon$	*List of parallel processes*
$\mathcal{M}$	$::= 0$	*Empty process*
	$\mid \lfloor k_i, \lambda_i : \mathcal{P} \rfloor \quad \mid \quad k_1, \lambda_1 : \mathcal{P}$	*(Frozen) process*
	$\mid [\mu; k_1]$	*Memory*
	$\mid \texttt{rl}k_1$	*Rollback token*
	$\mid k_1, \lambda_2 : \lambda_1[\mathcal{P}]$	*Message*
	$\mid k_1, \lambda_1 : \bot$	*Failed process*
$\mathcal{P}, \mathcal{Q}$	$::= 0 \quad \mid \quad x$	*Empty program/Variable*
	$\mid \lambda\langle\mathcal{P}\rangle \cdot \mathcal{Q} \quad \mid \quad x \triangleright_l \mathcal{P}$	*Send/Deliver message*
μ	$::= k_1, \lambda_2 : \lambda_1[\mathcal{P}] \parallel k_2, \lambda_2 : X \triangleright_\lambda \mathcal{Q}$	*Record*
	$\mid \lfloor k_1, \lambda_2 : \lambda_1[\mathcal{P}] \rfloor \parallel k_2, \lambda_2 : X \triangleright_\lambda \mathcal{Q}$	*Record with frozen message*
	$\mid k_1, \lambda_2 : \lambda_1[\mathcal{P}] \parallel \lfloor k_2, \lambda_2 : X \triangleright_\lambda \mathcal{Q} \rfloor$	*Record with frozen delivery*
	$\mid \lfloor k_1, \lambda_2 : \lambda_1[\mathcal{P}] \rfloor \parallel \lfloor k_2, \lambda_2 : X \triangleright_\lambda \mathcal{Q} \rfloor$	*Record fully frozen*

Fig. 5. lr-π syntax

3.2 Semantics

The semantics of the lr-π language is defined using a forward reduction relation (noted $\twoheadrightarrow$) and a backward reduction relation (noted $\rightsquigarrow$). Reduction rules are given in Figure **??**.

The forward reduction is defined by inference rules to send and deliver a message (L.SEND and L.DELIVER), to idle (L.IDLE), to fail (L.FAIL) and to lose a message (L.LOSE).

The backward reduction works in three steps. First, one needs to target a previous state to revert to: the rule L.ROLLBACK creates a rollback token ($\texttt{rl}k$) which indicates that the memory tagged k is to be restored[3]. The second step consists in tracking the causal dependency of the targeted memory: the span rule (L.SPAN) recursively freezes dependent processes; when all dependent processes are frozen, this step ends (rule L.TOP). The last steps consists in recursively restoring memories (rule L.DESCEND). Notice that, in lr-π, in contrast to SCL, a single rollback can affect multiple processes.

[3] For simplicity, we let this choice be non-deterministic, but we could easily extend the syntax of lr-π to accommodate e.g. imperative rollback instructions as in [**?**].

The L.SPAN rule actually comes in multiple flavours, depending on whether the message or the delivery process are already frozen. For the sake of conciseness, we only present one flavour where message and delivery process are not frozen.

$$\text{L.DELIVER}\ \frac{\mu = k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : X \rhd_l Q}{\begin{array}{r} k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : X \rhd_l Q \\ \twoheadrightarrow_d \mathtt{succ}(k_2), \lambda_2 : Q\{^{P,\lambda_1}/_{X,l}\} \parallel [\mu, \mathtt{succ}(k_2)] \end{array}}$$

$$\text{L.SEND}\ k_1, \lambda_1 : \lambda_2\langle P\rangle \cdot Q_1 \parallel k_2, \lambda_2 : Q_2 \twoheadrightarrow_s k_1, \lambda_1 : Q_1 \parallel k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Q_2$$

$$\text{L.IDLE}\ M \twoheadrightarrow_i M \qquad\qquad \text{L.FAIL}\ k_1, \lambda_1 : P \twoheadrightarrow_\perp k_1, \lambda_1 : \perp$$

$$\text{L.LOSE}\ k_1, \lambda_2 : \lambda_1[P] \parallel M \twoheadrightarrow_l M$$

$$\text{L.ROLLBACK}\ k_1, \lambda_1 : P \parallel [\mu, k_1] \rightsquigarrow_s k_1, \lambda_1 : P \parallel [\mu, k_1] \parallel \mathtt{rl}k_1$$

$$\text{L.DESCEND}\ \frac{M \text{ does not contain } k_1 \text{ labelled processes}}{\prod \lfloor k_1, \lambda_i : P_i \rfloor \parallel [\mu, k_1] \parallel M \rightsquigarrow_d \mu \parallel M}$$

$$\text{L.TOP}\ \frac{M \text{ does not contain } k_1 \text{ labelled processes}}{(\prod k_1, \lambda_i : P_i) \parallel M \parallel \mathtt{rl}k_1 \rightsquigarrow_t (\prod \lfloor k_1, \lambda_i : P_i \rfloor) \parallel M}$$

$$\text{L.SPAN}\ \frac{N \text{ and } M_i \text{ do not contain } k_1 \text{ labelled processes}}{\begin{array}{r} (\prod [k_1, \lambda_i : P_i \parallel M_i; k_i]) \parallel \prod k_1, \lambda_j : P_j \parallel \mathtt{rl}k_1 \parallel N \\ \rightsquigarrow_{sp} (\prod [\lfloor k_1, \lambda_i : P_i \rfloor \parallel M_i; k_i] \parallel \mathtt{rl}k_i) \parallel \prod \lfloor k_1, \lambda_j : P_j \rfloor \parallel N \end{array}}$$

Fig. 6. Reduction rules of lr-π

The example in Figure **??** shows two processes λ_1 and λ_2 exchanging a process P. λ_1 sends P to λ_2 and waits for an answer. When λ_2 receives the message, a memory is created and λ_2 sends back the message to λ_1 then executes P. When λ_1 receives the answer, a second memory is created.

The second part of the example shows the reverse execution of the previous exchanges. A rollback token $\mathtt{rl}k_2'$ is introduced. In a L.SPAN reduction, the process and the message tagged with k_2' are frozen, and a rollback roken $\mathtt{rl}k_1'$ is created. Then, the process tagged with k_1' is frozen in a L.TOP reduction. Finally, two L.DESCEND reductions reverse the configuration in the state preceding the first delivery of λ_2.

This example highlight a major difference from SCL: reversing λ_2 requires to reverse λ_1 in order to preserve the causal dependency. Contrary to SCL (and MANETHO), reversing a process is guarantee to only affect this process.

$$
\begin{aligned}
& k_1, \lambda_1 : \lambda_2\langle P\rangle \cdot X \triangleright_l X \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y) \\
\twoheadrightarrow_s\ & k_1, \lambda_1 : X \triangleright_l X \parallel k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y) \\
\twoheadrightarrow_d\ & k_1, \lambda_1 : X \triangleright_l X \parallel k_2', \lambda_2 : \lambda_1\langle P\rangle \cdot P \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \\
\twoheadrightarrow_s\ & k_1, \lambda_1 : X \triangleright_l X \parallel k_2', \lambda_2 : P \parallel k_2', \lambda_1 : \lambda_2[P] \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \\
\twoheadrightarrow_d\ & k_1', \lambda_1 : P \parallel k_2', \lambda_2 : P \parallel [k_2', \lambda_1 : \lambda_2[P] \parallel k_1, \lambda_1 : X \triangleright_l X; k_1'] \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \\
\rightsquigarrow_s\ & k_1', \lambda_1 : P \parallel k_2', \lambda_2 : P \parallel [k_2', \lambda_1 : \lambda_2[P] \parallel k_1, \lambda_1 : X \triangleright_l X; k_1'] \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \parallel \mathtt{rl} k_2' \\
\rightsquigarrow_{sp}\ & k_1', \lambda_1 : P \parallel \lfloor k_2', \lambda_2 : P \rfloor \parallel [\lfloor k_2', \lambda_1 : \lambda_2[P] \rfloor \parallel k_1, \lambda_1 : X \triangleright_l X; k_1'] \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \parallel \mathtt{rl} k_1' \\
\rightsquigarrow_t\ & \lfloor k_1', \lambda_1 : P \rfloor \parallel \lfloor k_2', \lambda_2 : P \rfloor \parallel [\lfloor k_2', \lambda_1 : \lambda_2[P] \rfloor \parallel k_1, \lambda_1 : X \triangleright_l X; k_1'] \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \\
\rightsquigarrow_d\ & \lfloor k_2', \lambda_2 : P \rfloor \parallel \lfloor k_2', \lambda_1 : \lambda_2[P] \rfloor \parallel k_1, \lambda_1 : X \triangleright_l X \\
& \quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y); k_2'] \\
\rightsquigarrow_d\ & k_1, \lambda_1 : X \triangleright_l X \parallel k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y\rangle \cdot Y)
\end{aligned}
$$

Fig. 7. Example of lr-π forward and backward execution

3.3 Starting and Correct Configurations

Among all lr-π configurations, we distinguish correct and starting configurations.

A configuration is correct when (1) all variables and location variables are bound; (2) there is a single process for each λ_i; and (3) for each process $k, \lambda : \mathcal{P}$, for each $k_0 < k_i \leq k$, there is a memory tagged with k_i. A starting configuration is a correct configuration such that (1) there is no memory; (2) there is no pending message; (3) there is no frozen memory or frozen process; and (4) there is no rollback token.

The set of correct configurations is written $\mathbb{C}_{\text{lr-}\pi}$ and the set of starting configurations is written $\mathbb{C}^s_{\text{lr-}\pi}$.

3.4 Encoding in roll-π

The lr-π language is inspired by roll-π described in [**?**]. Similarly to lr-π, $\mathbb{C}_{\text{roll-}\pi}$ denotes the set of correct roll-π configurations. In order to show it inherits the causal consistency property of roll-π, we sketch in this section, how lr-π can be encoded into roll-π (full details available in [**?**]). The major difference between the two languages is that lr-π allows failure, which roll-π doesn't, and that roll-π message sending is without continuation. A few minor subtleties also occurs: implementing the loss of message, an idle process and the introduction of `roll` tokens, which can all be encoded very simply.

Process Failure We want to be able to stop each process at any point. Given a process P, we encode it as $\nu I \cdot I(X) \triangleright X \parallel I\langle 0\rangle \parallel I\langle P\rangle$. Hence, it reduces either to $\nu I \cdot P \parallel I\langle 0\rangle$ (the $I\langle 0\rangle$ part being garbage), either to $\nu I \cdot 0 \parallel I\langle P\rangle$, with $I\langle P\rangle$ being blocked (memory creation and tag changes not shown).

By applying this strategy recursively to P, it is possible to block the execution at each step. The only way to unblock the process is to revert it, which is exactly what a failure does in lr-π.

Thus, we define $\delta_{\cdot}(\cdot)$ which creates the failure machinery as follow:

$$\delta_\lambda(P) = \nu I \cdot I(X) \triangleright X \parallel I\langle 0\rangle \parallel I\langle \mathtt{tr}_\lambda(P)\rangle \tag{1}$$

with $\mathtt{tr}(\cdot)$ translating lr-π programs into roll-π ones.

Message Loss In the original roll-π, messages cannot be lost. To encode message loss, we simply add a consumer process.

4 Simulation of scl by lr-π

4.1 Translation from scl to lr-π

We now define the function γ which translates an SCL configuration into an lr-π one. Most of this translation is intuitive, only creating memories is not trivial and the intuition is given below. The long version of the paper contains details of the translation.

Given an SCL configuration $M = \langle T_1 \parallel \ldots \parallel T_n, \mathcal{L}, \mathcal{P}\rangle$ we define

$$\gamma(M) = \gamma_T(T_1) \parallel \ldots \parallel \gamma_T(T_n) \parallel \gamma_L(\mathcal{L}) \parallel \gamma_M(M) \tag{2}$$

where γ_T trivially translates the SCL process into its lr-π equivalent (and defaults to $\bot$ if the process is marked or $\bot$). γ_L recreates an lr-π pending message for each message in a list. Finally γ_M recreates memories according to the idea below.

Managing Memories Given an SCL configuration, we can infer the last step of a process k (and thus the previous state of the configuration):

- if there is a pending message m sent by k such that the antecedence graph piggybacked is the current antecedence graph of k and that the *ssn* of k is the successor of the *ssn* piggybacked in m, then the last step of k was to send m.
- otherwise, if the $\mathcal{AG}$ of k is not a single node, then the last step was to receive a message. The message that triggered the current state interval can be retrieved from the antecedence graph.
- finally, if the antecedence graph of k is a single node, then k is in its initial state.

Hence, by applying recursively the above idea, one can infer the full history of a given process k and then, each time a message delivery is matched, create the corresponding memory in lr-π.

4.2 Simulation

We now show that, for any SCL configuration M, its encoding $\gamma(M)$ in lr-π faithfully simulates M. In both SCL and lr-π, an observable is a message targeted to some process denoted by its identifier d (noted $M \downarrow_d$).

Definition 3 (Barbed bisimulation). *A relation $\mathcal{R} \subseteq \mathbb{C}_{\text{SCL}} \times \mathbb{C}_{lr\text{-}\pi}$ is a strong (resp. weak) barbed simulation if whenever $(M, N) \in \mathcal{R}$*

- *$M \downarrow_d$ implies $N \downarrow_d$ (resp. $N \rightarrow^\star \downarrow_d$)*
- *$M \rightarrow M'$ implies $N \rightarrow N'$ (resp. $N \rightarrow^\star N'$) with $M' \mathcal{R} N'$*

$\mathcal{R}$ is a strong (resp. weak) barbed bisimulation if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong (resp. weak) barbed simulations.

Theorem 2 (Simulation). *The relation $\mathcal{R} = \{(M, \gamma(M)) | M \in \mathbb{C}_{\text{SCL}}\}$ is a weak barbed simulation.*

The proof, which is detailed in [**?**], relies on the two following lemmas.

Lemma 1 (Observable conservation). *If*

$$(M, \gamma(M)) \in \mathcal{R} \tag{3}$$

then

$$M \downarrow_d \Leftrightarrow \gamma(M) \downarrow_d \tag{4}$$

The lemma is a simple consequence of the definition of *observable* and of the encoding function $\gamma(\cdot)$.

Lemma 2 ($\mathcal{R}$ closure under reduction). *Assuming $M \in \mathbb{C}_{\text{SCL}}$ and $M_{lr\text{-}\pi} = \gamma(M)$ (i.e. $(M, M_{lr\text{-}\pi}) \in \mathcal{R}$).*

If $M \rightarrow M'$ then exists a sequence of lr-π reductions $\rightarrow^\star_{lr\text{-}\pi}$ such that

$$M_{lr\text{-}\pi} \rightarrow^\star_{lr\text{-}\pi} M'_{lr\text{-}\pi} \tag{5}$$

and

$$(M', M'_{lr\text{-}\pi}) \in \mathcal{R} \tag{6}$$

The proof of this lemma proceeds by case study on the SCL reduction. Most cases are trivial, in particular, all backward rules except S.REPLAY.STOP are either L.LOSE or L.IDLE. The S.FAIL rule matches L.FAIL. The forward rules are verbose, but direct. Only the S.REPLAY.STOP case involves some complication. To prove it, we first show that for any SCL recovery sequence, there exists a corresponding lr-π one. Since none of the other SCL backward reductions match the lr-π, then S.REPLAY.STOP does.

4.3 Discussion on bisimulation

The above relation $\mathcal{R}$ is a simulation but is not a bisimulation. In SCL, once a process takes a checkpoint, the process can not rollback before this checkpoint. To tackle this difference, we define a weak version of lr-π called lr-π^- with constraints on the rollback point and on message loss.

The lr-π^- Language In MANETHO, when a single process fails, it rolls back to its last public state, i.e. to the last state in which it sent a message which has been received. In lr-π, when such a message is delivered, the receiver creates a memory, hence, we constrain the rollback of a process to the last state such that exists a memory with a message tagged with this state.

Furthermore, we know that all messages sent during and before the target state interval are ignored, hence, in lr-π^-, these are marked and ignored.

$$\begin{array}{lll} \mathcal{M} ::= \ldots & & \textit{Same than lr-}\pi \\ \quad \mid [\mu; k_1]^\bullet & & \textit{Marked memory} \\ \quad \mid k_1, \lambda_2 : \lambda_1[\mathcal{P}]^\bullet & & \textit{Marked message} \end{array}$$

Fig. 8. lr-π^- syntax, modifications from lr-π syntax

The forward semantics of lr-π^- are the same than lr-π, only the backward rules change. Starting configurations of lr-π^- are the same than lr-π, and correct configurations ($\mathbb{C}_{\text{lr-}\pi^-}$) are defined analogously.

The rollback start is modified to restrict rollback targets: let k_j^i be the biggest $k^i \in K_{\lambda_i}$ such that there exists $[P \parallel k_j^i, \lambda : \lambda_i[Q]; k] \in M$ (or $k_j^i = k_0^i$ if none exists). This memory ensures that the state k_j^i is the last state known by an other process. If there exists a memory containing a process tagged with k_j^i, we mark that memory using LM.START, as well as all pending messages sent by the process being reverted:

$$\frac{\mu = k_j^i, \lambda_i : P \parallel k, \lambda : \lambda_i[Q]}{M' \parallel [\mu; k'] \parallel \prod k, \lambda : \lambda_i[P] \rightsquigarrow_s M' \parallel [\mu; k']^\bullet \parallel \prod k, \lambda : \lambda_i[P]^\bullet \parallel \texttt{rl}k'}$$

If no such memory exists, we simply fast forward toward the end of the current state interval with LM.FORWARD:

$$M' \parallel k_j^i, \lambda_i : P \parallel \prod k, \lambda : \lambda_i[P] \rightsquigarrow_f M' \parallel k_j^i, \lambda_i : \theta(P) \parallel \prod k, \lambda : \lambda_i[P]^\bullet$$

where the function θ simulates the local replay of messages:

$$\theta(P) = \begin{cases} \theta(k, \lambda, \texttt{succ}(r) : Q) & \text{if } P = k, \lambda, r : \lambda_1\langle R\rangle \cdot Q \\ P & \text{otherwise} \end{cases}$$

Marked messages are messages to be ignored, hence only these can be removed: LM.IGNORE replaces L.LOSE:

$$k_1, \lambda_2 : \lambda_1[P]^\bullet \parallel M \twoheadrightarrow_l M$$

The rollback semantics consists in reverting back to a marked memory, then replaying all send actions locally:

$$\text{LM.ROLL}\ \frac{N \blacktriangleright k \qquad \texttt{complete}(N \parallel [\mu; k]) \qquad \mu = k_1, \lambda_1 : P \parallel k_2, \lambda_1 : \lambda_2[Q]}{M \parallel N \parallel [\mu; k]^\bullet \parallel \texttt{rl}k \rightsquigarrow_r M \parallel k_1, \lambda_1 : P}$$

with $\blacktriangleright$ and `complete` defined similarly to [**?**]:

Definition 4 (Causal dependence). *Let M be a configuration and T_M be the set of tags in M. We define the relation $>_M$ on T_M as the smallest relation satisfying: $k' >_M k$ if k' occurs in μ for some (marked) memory $[\mu; k]$ that occurs in M. The causal dependence relation $:>$ is the reflexive and transitive closure of $>_M$.*

Definition 5 (k-dependence). *Given $M = \prod_{i \in I} k_i, \lambda_i : P_i \parallel \prod_{j \in J} k_j, \lambda_d : \lambda_s[P_j] \parallel \prod_{l \in L}[\mu_l; k_l]$. M is k-dependent (noted $M \blacktriangleright k$) if $\forall i \in I \cup J \cup L \cdot k :> k_i$.*

Definition 6 (Complete configuration). *An lr-π^- configuration M is complete (noted $\texttt{complete}(M)$) if, for each memory $[\mu; k]$ in M, there exists a process $k, \lambda : P$ or a memory $[k, \lambda : P \parallel Q; k']$ in M.*

Unlike lr-π, reversing a memory is done in a single LM.ROLL step in lr-π^-. Since, as explained above, the marked memory in lr-π^- corresponds to the state interval SCL rolls back to, SCL can simulate the LM.ROLL and LM.FORWARD rules with a complete rollback sequence. Also, LM.IGNORE is simulated by S.LOSE. Finally, the rule LM.START simply adds marks and is simulated by S.IDLE.

Figure **??** shows an example of lr-π^- backward execution. The initial configuration is the same than in Figure **??**. Since forward rules are the same in both languages, we only show the backward reduction.

$$\begin{aligned}
&k_1', \lambda_1 : P \parallel k_2', \lambda_2 : P \parallel [k_2', \lambda_1 : \lambda_2[P] \parallel k_1, \lambda_1 : X \triangleright_l X; k_1'] \\
&\quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y \rangle \cdot Y); k_2'] \\
\rightsquigarrow_s\ &k_1', \lambda_1 : P \parallel k_2', \lambda_2 : P \parallel [k_2', \lambda_1 : \lambda_2[P] \parallel k_1, \lambda_1 : X \triangleright_l X; k_1']^\bullet \parallel \texttt{rl}k_1' \\
&\quad \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y \rangle \cdot Y); k_2'] \\
\rightsquigarrow_r\ &k_2', \lambda_2 : P \parallel k_1, \lambda_1 : X \triangleright_l X \parallel [k_1, \lambda_2 : \lambda_1[P] \parallel k_2, \lambda_2 : Y \triangleright_m (m\langle Y \rangle \cdot Y); k_2']
\end{aligned}$$

Fig. 9. Example of lr-π^- backward execution. In this example, the process λ_1 is reverted.

In order to study the simulation of lr-π^-, we define observable messages:

Definition 7 (Observable in lr-π^-). *In a lr-π^- configuration $M_{lr\text{-}\pi^-}$, λ_2 is observable (noted $M_{lr\text{-}\pi^-} \downarrow_{\lambda_2}$) if and only if $M_{lr\text{-}\pi^-} \equiv k, \lambda_2 : \lambda_1[Q] \parallel M$.*

Notice that marked messages are not observable. We refine the definition of observable message in SCL:

Definition 8 (Observable in scl (2)). *In a* SCL *configuration* M_{SCL}*, d is observable (noted* $M_{\text{SCL}} \downarrow_d$*) if and only if there exists a* $\langle k, ssn, \mathcal{AG}, si, Q, inc\rangle \in {}_sL_d$ *and the incarnation number of k is inc.*

This definition only differs from the first one in the way that only messages which have been sent since the last rollback are observable.

Theorem 3 (Simulation). *The relation* $\mathcal{R} = \{(M, \gamma_+^{-1}(M)) | M \in \mathbb{C}_{lr\text{-}\pi^-}\}$ *is a weak barbed simulation.*

The proof is similar to the proof of Theorem **??** and is provided in [**?**].

5 Related Work

We do not know of any work that attempts to relate a distributed checkpoint rollback scheme with a reversible model of computation, as we do in this paper. However, our work touches upon several topics, including the formal specification and verification of distributed checkpoint rollback algorithms, the definition of rollback and recovery primitives and abstractions in concurrent programming languages and models, and the study of reversible programming models and languages. We discuss these connections below.

Several works consider the correctness of distributed checkpoint rollback algorithms. The MANETHO algorithm was introduced informally using pseudo code and proved correct in [**?**,**?**]. Several other checkpointing algorithms have been considered and proved correct, such as e.g. adaptive checkpointing [**?**] or checkpointing with mutable checkpoints [**?**]. Our proof of correctness for MANETHO relies on a more formal presentation of the algorithm, by way of operational semantics rules, and the analysis of the associated transition relation. In that respect, the work which seems closer to ours is [**?**], which also formalizes a checkpointing algorithm (the algorithm from [**?**], which is not a causal logging checkpoint/rollback algorithm) by means of operational semantics rules, and also proves its correctness by an inductive analysis of its transition relation.

The rollback capability in lr-π is directly derived from the low-level semantics of roll-π [**?**]. Compared to [**?**], the rollback capability in lr-π can be triggered by the occurrence of a process crash, but we have shown above that this could be encoded in roll-π. Undo or rollback capabilities in programming languages have a long history (see e.g. [**?**] for an early survey in sequential languages). More recent works which have introduced undo or rollback capabilities in a concurrent programming language or model include [**?**], which defines logging and group primitives for programming fault-tolerant systems, [**?**], which extends the actor model of computation with primitives for creating globally-consistent checkpoints, [**?**], which introduces checkpointing primitives in concurrent ML, [**?**], which extends the KLAIM tuple space programming language with rollback capabilities directly inspired by roll-π, and [**?**] which extends a subset of *Core*

Erlang with a reversible semantics similar to the roll-π one. The rollback capabilities in roll-π have several advantages over these different works: rollback is possible at any moment during execution, in contrast to [**?**]; does not suffer from the domino effect, in contrast to [**?**]; and provides direct support for consistently undoing all the consequences of a given action, in contrast to [**?**]. The same properties hold for lr-π and reversible KLAIM.

6 Conclusion

We have shown in this paper the tight relationship that exists between a checkpoint/rollback scheme based on causal logging and a reversible concurrent programming model based on causal consistency. More precisely, we have shown that the SCL language, whose operational semantics formalizes the behaviour of the MANETHO algorithm, can be (weakly barbed) simulated by lr-π, a reversible asynchronous concurrent language with process crash failures, based on the roll-π language, a reversible π-calculus with explicit rollbacks. The converse is not true, but we have shown that SCL can (weakly barbed) simulate a variant of lr-π with limited rollbacks. These results probably extend to other checkpoint/rollback schemes based on causal logging, but one would need first to formally specify them as we did in this paper for MANETHO.

Apart from showing this relationship, the results are interesting for several reasons. On the one hand, they point to interesting extensions to causal logging checkpoint/rollback schemes. In effect lr-π constitutes an extension of checkpoint/rollback causal logging that does not limit rollbacks to the last saved checkpoint of a failed process: this can be a useful feature to protect against catastrophic faults such as those resulting from faulty upgrades. Also, it is trivial to add to lr-π the ability to create new processes and to exchange processes in messages as in roll-π, thus extending checkpoint/rollback capabilities to dynamic environments, where new code can be added and new processes can be created at runtime, or to add compensation capabilities as in [**?**] to avoid retrying a faulty execution path. We do not know of checkpoint/rollback schemes that combine these different capabilities and the tight connection established in this paper shows with lr-π how they can be added to causal logging checkpoint/rollback schemes. On the other hand, they suggest interesting directions to optimize rollback in reversible concurrent languages. For instance, as in MANETHO, one can avoid rolling back all processes in lr-π by a judicious use of local replay.